Parallel Frontal Solver for Large-Scale Process Simulation and Optimization

J. U. Mallya, S. E. Zitney, and S. Choudhary
Cray Research, Inc., Eagan, MN 55121

M. A. Stadtherr
Dept. of Chemical Engineering, University of Notre Dame, Notre Dame, IN 46556

For the simulation and optimization of large-scale chemical processes, the overall computing time is often dominated by the time needed to solve a large sparse system of linear equations. A new parallel frontal solver that can significantly reduce the wallclock time required to solve these linear equation systems using parallel/vector supercomputers is presented. The algorithm exploits both multiprocessing and vector processing by using a multilevel approach in which frontal elimination is used for the partial factorization of each front. Results of several large-scale process simulation and optimization problems are presented.

Introduction

The solution of realistic, industrial-scale simulation and optimization problems is computationally very intense, and may require the use of high-performance computing technology to be done in a timely manner. For example, Zitney et al. (1995) described a dynamic simulation problem at Bayer AG requiring 18 hours of CPU time on a CRAY C90 supercomputer when solved with the standard implementation of SPEEDUP (Aspen Technology, Inc.). To better use this leading-edge technology in process simulation and optimization requires the use of techniques that efficiently exploit vector and parallel processing. Since most current techniques were developed for use on conventional series machines, it is often necessary to rethink problem-solving strategies in order to take full advantage of supercomputing power. For example, by using a different linear equation-solving algorithm and addressing other implementation issues, Zitney et al. (1995) reduced the time needed to solve the Bayer problem from 18 hours to 21 minutes.

In the Bayer problem, as in most other industrial-scale problems, the solution of large, sparse systems of linear equations is the single most computationally intensive step, requiring over 80% of the total simulation time in some cases. Thus, any reduction in the linear system solution time will result in a significant reduction in the total simulation time for a given problem, as well as provide the potential for solving much larger problems within a given time frame. The matrices that arise, however, do not have any of the desirable properties, such as numerical or structural symmetry, positive definiteness, and bandedness often associated with sparse matrices, and usually exploited in developing efficient parallel/vector algorithms. Recently, an implementation of the frontal method (Zitney, 1992; Zitney and Stadtherr, 1993; Zitney et al., 1995), developed at the University of Illinois and later extended at Cray Research, Inc., has been described that is designed specifically for use in the context of process simulation. This solver (FAMP) has been incorporated in CRAY implementations of popular commercial codes, such as ASPEN PLUS, SPEEDUP (Aspen Technology, Inc.), and NOVA (Dynamic Optimization Technology Products, Inc.). FAMP is effective on vector machines since most of the computations involved can be performed using efficiently vectorized dense matrix kernels. However, this solver does not well exploit the multiprocessing architecture of parallel/vector supercomputers. In this article we propose a new parallel/vector frontal solver (PFAMP) that exploits both the vector *and* parallel-processing architectures of modern supercomputers. Results demonstrate that the approach described leads to significant reductions in the wallclock time required to solve the sparse linear systems arising in large-scale process simulation and optimization.

Correspondence concerning this article should be addressed to M. A. Stadtherr.
Current address of S. E. Zitney: AspenTech UK Ltd., Castle Park, Cambridge CB3 0AX, England.

Background

Consider the solution of a linear equation system $Ax = b$, where A is a large sparse $n \times n$ matrix and x and b are column vectors of length n. While iterative methods can be used to solve such systems, the reliability of such methods is questionable in the context of process simulation (Cofer and Stadtherr, 1996). Thus we concentrate here on direct methods. Generally such methods can be interpreted as an LU factorization scheme in which A is factored $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix. Thus, $Ax = (LU)x = L(Ux) = b$, and the system can be solved by a simple forward substitution to solve $Ly = b$ for y, followed by a back substitution to find the solution vector x from $Ux = y$.

The frontal elimination scheme used here is an LU factorization technique that was originally developed to solve the banded matrices arising in finite-element problems (Irons, 1970; Hood, 1976). The original motivation was, by limiting computational work to a relatively small *frontal matrix*, to be able to solve problems on machines with small core memories. Today it is widely used for finite-element problems on vector supercomputers because, since the frontal matrix can be treated as dense, most of the computations involved can be performed by using very efficient vectorized dense matrix kernels. Stadtherr and Vegeais (1985) extended this idea to the solution of process simulation problems on supercomputers, and later (Vegeais and Stadtherr, 1990) demonstrated its potential. As noted earlier, an implementation of the frontal method developed specifically for use in the process-simulation context has been described by Zitney (1992), Zitney and Stadtherr (1993), and Zitney et al. (1995), and is now incorporated in supercomputer versions of popular process simulation and optimization codes.

The frontal elimination scheme can be outlined briefly as follows:

1. Assemble a row into the frontal matrix.
2. Determine if any columns are fully summed in the frontal matrix. A column is fully summed if it has all of its nonzero elements in the frontal matrix.
3. If there are fully summed columns, then perform partial pivoting in those columns, eliminating the pivot rows and columns and doing an outer-product update on the remaining part of the frontal matrix.

This procedure begins with the assembly of row 1 into the initially empty frontal matrix, and proceeds sequentially row by row until all are eliminated, thus completing the LU factorization. To be more precise, it is the LU factors of the permuted matrix PAQ that have been found, where P is a row permutation matrix determined by the partial pivoting, and Q is a column permutation matrix determined by the order in which the columns become fully summed. Thus the solution to $Ax = b$ is found as the solution to the equivalent system $PAQQ^Tx = LUQ^Tx = Pb$, which is solved by forward substitution to solve $Ly = Pb$ for y, back substitution to solve $Uw = y$ for w, and finally the permutation $x = Qw$. To simplify notation, the permutation matrices will henceforth not be shown explicitly.

To see this in mathematical terms, consider the submatrix $A^{(k)}$ remaining to be factored after the $(k-1)$th pivot:

$$A^{(k)} = \left[\begin{array}{c|c} F^{(k)} & 0 \\ \hline A_{ps}^{(k)} & A_{ns}^{(k)} \end{array}\right]. \tag{1}$$

Here $F^{(k)}$ is the frontal matrix. The subscript ps in $A_{ps}^{(k)}$ indicates that it contains columns that are partially summed (some but not all nonzeros in the frontal matrix), and the subscript ns in $A_{ns}^{(k)}$ indicates that it contains columns that are not summed (no nonzeros in the frontal matrix). If a stage in the elimination process has been reached at which all remaining columns have nonzeros in the frontal matrix, then $A_{ns}^{(k)}$ and the corresponding zero submatrix will not appear in Eq. 1. Assembly of rows into the frontal matrix then proceeds until $g_k \geq 1$ columns become fully summed:

$$A^{(k)} = \left[\begin{array}{c|c|c} \bar{F}_{11}^{(k)} & \bar{F}_{12}^{(k)} & 0 \\ \hline \bar{F}_{21}^{(k)} & \bar{F}_{22}^{(k)} & 0 \\ \hline 0 & \bar{A}_{ps}^{(k)} & \bar{A}_{ns}^{(k)} \end{array}\right], \tag{2}$$

where $\bar{F}^{(k)}$ is now the frontal matrix, and its submatrices $\bar{F}_{11}^{(k)}$ and $\bar{F}_{21}^{(k)}$ comprise the columns that have become fully summed, which are now eliminated using rows chosen during partial pivoting and which are shown as belonging to $\bar{F}_{11}^{(k)}$ here. This amounts to the factorization $\bar{F}_{11}^{(k)} = L_{11}^{(k)}U_{11}^{(k)}$ of the order-g_k block $\bar{F}_{11}^{(k)}$, resulting in

$$A^{(k)} = \left[\begin{array}{c|c|c} L_{11}^{(k)}U_{11}^{(k)} & U_{12}^{(k)} & 0 \\ \hline L_{21}^{(k)} & F^{(k+g_k)} & 0 \\ \hline 0 & A_{ps}^{(k+g_k)} & A_{ns}^{(k+g_k)} \end{array}\right], \tag{3}$$

where the new frontal matrix $F^{(k+g_k)}$ is the Schur complement $F^{(k+g_k)} = \bar{F}_{22}^{(k)} - L_{21}^{(k)}U_{12}^{(k)}$, which is computed using an efficient full-matrix outer-product update kernel, $A_{ps}^{(k+g_k)} = \bar{A}_{ps}^{(k)}$ and $A_{ns}^{(k+g_k)} = \bar{A}_{ns}^{(k)}$. Note that operations are done within the frontal matrix only. At this point $L_{11}^{(k)}$ and $L_{21}^{(k)}$ contain columns k through $k + g_k - 1$ of L, and $U_{11}^{(k)}$ and $U_{12}^{(k)}$ contain rows k through $k + g_k - 1$ of U. The computed columns of L and rows of U are saved and the procedure continues with the assembly of the next row into the new frontal matrix $F^{(k+g_k)}$. Note that the index k used for the $A^{(k)}$ submatrix and its blocks occurs in increments of g_k. That is, after $A^{(k)}$ is processed, the next submatrix to be processed is $A^{(k+g_k)}$. Thus, since g_k may be greater than one, $A^{(k)}$ may not occur for all integers k.

Small-Grained Parallelism

In the frontal algorithm, the most expensive stage computationally is the outer-product update of the frontal matrices. When executed on a single vector processor, FAMP performs efficiently because the outer-product update is readily vectorized. Essentially this takes advantage of a fine-grained, machine-level parallelism, in this case (CRAY C90) the overlapping, assembly-line-style parallelism provided by the internal architecture of a highly pipelined vector processor. An additional level of parallelism might be obtained by *multitasking* the innermost loops that perform the outer-product update. Multitasking refers to the use of multiple processors to execute a task with the aim of decreasing the actual wall-clock time for that task. *Microtasking*, which is a form of multitasking, involves the processing of tasks with small granularity. Typically, these independent tasks can be identified quickly and exploited without rewriting large portions of the code. Specific directives in the source code control microtasking by designating the bounds of a control structure in which each iteration of a DO loop is a process that can be executed in parallel. Since there is an implicit synchronization point at the bottom of every control structure, all iterations within the control structure must be finished before the processors can proceed. Our experience (Mallya, 1996) shows that the potential for achieving good speedups in FAMP with the exploitation of only small-grained parallelism by microtasking is very limited. Often, speedups of only about 1.2 could be achieved using four processors. The reason is that parallel tasks are simply not computationally intense enough to overcome the synchronization cost and the overhead associated with invoking multiple processors on the C90. This indicates the need for exploiting a higher, coarse-grained level of parallelism to make multiprocessing worthwhile for the solution of sparse linear systems in process simulation and optimization.

Coarse-Grained Parallelism

The main deficiency with the frontal-code FAMP is that there is little opportunity for parallelism beyond that which can be achieved by microtasking the inner loops or by using higher level BLAS in performing the outer product update (Mallya, 1996; Camarda and Stadtherr, 1993). We overcome this problem by using a coarse-grained parallel approach in which frontal elimination is performed simultaneously in multiple independent or loosely connected blocks. This can be interpreted as applying frontal elimination to the diagonal blocks in a bordered block-diagonal matrix.

Consider a matrix in singly bordered block-diagonal form:

$$A = \left[\begin{array}{cccc} A_{11} & & & \\ & A_{22} & & \\ & & \ddots & \\ & & & A_{NN} \\ \hline S_1 & S_2 & \cdots & S_N \end{array}\right], \tag{4}$$

where the diagonal blocks A_{ii} are $m_i \times n_i$ and in general are rectangular with $n_i \geq m_i$. Because of the unit-stream nature of the problem, process simulation matrices occur naturally in this form, as described in detail by Westerberg and Berna (1978). Each diagonal block A_{ii} comprises the model equations for a particular unit, and equations describing the connections between units, together with design specifications, constitute the border (the S_i). Of course, not all process simulation codes may use this type of problem formulation, or order the matrix directly into this form. Thus some matrix reordering scheme may need to be applied, as discussed further below.

The basic idea in the parallel frontal algorithm (PFAMP) is to use frontal elimination to partially factor each of the A_{ii}, with each such task assigned to a separate processor. Since the A_{ii} are rectangular in general, it usually will not be possible to eliminate all the variables in the block, nor perhaps, for numerical reasons, all the equations in the block. The equations and variables that remain, together with the border equations, form a "reduced" or "interface" matrix that must then be factored.

The parallel frontal solver can also be interpreted as a special case of the *multifrontal* approach. The multifrontal method (Duff and Reid, 1983, 1984; Duff, 1986; Amestoy and Duff, 1989) is a generalization of the frontal method that has also been widely applied (e.g., Montry et al., 1987; Lucas et al., 1987; Ingle and Mountziaris, 1995; Zitney et al., 1996; Mallya and Stadtherr, 1997) in the context of parallel and/or vector computing. In the classic multifrontal method, which is most useful for symmetric or nearly symmetric systems, an elimination tree is used to represent the order in which a series of frontal matrices is partially factored, and to determine which of these partial factorizations can be performed in parallel. In the more general unsymmetric-pattern multifrontal method (Davis and Duff, 1997), the elimination tree is replaced by a directed acyclic graph. In the conventional multifrontal method, whether the classic or unsymmetric-pattern approach, there is generally a relatively large number of relatively small frontal matrices, only some of which can be partially factored in parallel, and the partial factorizations are done by conventional elimination. In the parallel frontal method described here, there is generally a relatively small number of relatively large frontal matrices, all of which are partially factored in parallel, and the partial factorizations are done by frontal elimination. No elimination tree or directed acyclic graph is required. Duff and Scott (1994) have applied this type of approach in solving finite-element problems and referred to it as a "multiple-fronts" approach in order to distinguish it from the conventional multifrontal approach. The parallel frontal solver described here represents a multilevel approach to parallelism in which, at the lower level, frontal elimination is used to exploit the fine-grained parallelism available in a single vector processor, while, at the upper level, there is a coarse-grained distribution of tasks across multiple vector processors.

PFAMP algorithm

The basic PFAMP algorithm is outlined below, followed by a more detailed explanation of the key steps.

Algorithm PFAMP

Begin parallel computation on P processors

For $i = 1{:}N$, with each task i assigned to the next available processor:

1. Do symbolic analysis on the diagonal block A_{ii} and the corresponding portion of the border (S_i) to obtain memory requirements and last-occurrence information (for determining when a column is fully summed) in preparation for frontal elimination.
2. Assemble nonzero rows of S_i into the frontal matrix.
3. Perform frontal elimination on A_{ii}, beginning with the assembly of the first row of A_{ii} into the frontal matrix (see the background section). The maximum number of variables that can be eliminated is m_i, but the actual number of pivots

done is $p_i \leq m_i$. The pivoting scheme used is described in detail below.

4. Store the computed columns of L and rows of U. Store the rows and columns remaining in the frontal matrix for assembly into the interface matrix.

End parallel computation

5. Assemble the interface matrix from the contributions of Step 4 and factor.

Note that for each block the result of Step 3 is

$$\begin{array}{c|cc} & C_i & C_i' \\ \hline R_i & L_iU_i & U_i' \\ R_i' & L_i' & F_i \end{array}, \tag{5}$$

where R_i and C_i are index sets comprising the p_i pivot rows and p_i pivot columns, respectively; R_i is a subset of the row index set of A_{ii}; R_i' contains row indices from S_i (the nonzero rows) as well as from any rows of A_{ii} that could not be eliminated for numerical reasons. As they are computed during Step 3, the computed columns of L and rows of U are saved in arrays local to each processor. Once the partial factorization of A_{ii} is complete, the computed block column of L and block row of U are written into global arrays in Step 4 before that processor is made available to start the factorization of another diagonal block. The remaining frontal matrix F_i is a contribution block that is stored in central memory for eventual assembly into the interface matrix in Step 5.

The overall situation at the end of the parallel computation section is

$$\begin{array}{c|cccc|c} & C_1 & C_2 & \cdots & C_N & C' \\ \hline R_1 & L_1U_1 & & & & U_1' \\ R_2 & & L_2U_2 & & & U_2' \\ \vdots & & & \ddots & & \vdots \\ R_N & & & & L_NU_N & U_N' \\ \hline R' & L_1' & L_2' & \cdots & L_N' & F \end{array}, \tag{6}$$

where

$$R' = \bigcup_{i=1}^{N} R_i' \quad \text{and} \quad C' = \bigcup_{i=1}^{N} C_i';$$

and F is the interface matrix that can be assembled from the contribution blocks F_i. Note that, since a row index in R' may appear in more than one of the R_i' and a column index in C' may appear in more than one of the C_i', some elements of F may get contributions from more than one of the F_i. As this doubly bordered block-diagonal form makes clear, once values of the variables in the interface problem have been solved for, the remaining triangular solves needed to complete the solution can be done in parallel using the same decomposition used to do the parallel frontal elimination. During this process the solution to the interface problem is made globally available to each processor.

Once factorization of all diagonal blocks is complete, the interface matrix is factored. This is carried out by using the FAMP solver, with microtasking to exploit loop-level parallelism for the outer-product update of the frontal matrix. However, as noted earlier, this tends to provide little speedup, so that the factorization of the interface problem can in most cases be regarded as essentially serial. It should also be noted that depending on the size and sparsity of the interface matrix, some solver other than FAMP may in fact be more attractive for performing the factorization. For example, if the order of the interface matrix is small, around 100 or less, then the use of a dense matrix factorization routine might be more appropriate.

Performance issues

In the parallel frontal algorithm, the computational work is done in two main phases: the factorization of the diagonal blocks and the factorization of the interface matrix. For the first phase, the performance of the algorithm will depend on how well the load can be balanced across the processors. Ideally, each diagonal block would be the same size (or more precisely, each would require the same amount of work to factor). In this ideal case, if there were $N = P$ diagonal blocks, then the speedup of this phase of the algorithm would scale linearly up to P processors (i.e., speedup of P on P processors). While this phase of the algorithm can be highly scalable, the second phase is likely to be much less so. Despite efforts to exploit small-grained parallelism in the factorization of the interface matrix, in most cases it remains essentially serial. A lesson of Amdahl's law is that even a small serial component in a computation can greatly limit scalability beyond a small number of processors [see Vegeais and Stadtherr (1992) for further discussion of this point]. Thus, the factorization of the interface matrix is a bottleneck and it is critical to keep the size of the interface problem small to achieve good speedups for the overall solution process.

Numerical pivoting

It is necessary to perform numerical pivoting to maintain stability during the elimination process. The frontal code FAMP uses partial pivoting to provide numerical stability. However, with the parallel frontal scheme of PFAMP, we need to ensure that the pivot row belongs to the diagonal block A_{ii}. We cannot pick a pivot row from the border S_i because border rows may be shared by more than one diagonal block. Thus for use here we propose a partial-threshold pivoting strategy. Partial pivoting is carried out to find the largest element in the pivot column while limiting the search to the rows that belong to the diagonal block A_{ii}. This element is chosen as the pivot element if it satisfies a threshold pivot tolerance criterion with respect to the largest element in the entire pivot column (including the rows that belong to the diagonal block A_{ii} and the border S_i). In order to explain this pivot strategy consider the example shown in Figure 1. Here rows 1–2 belong to the border S_i and rows 3–6 belong to the diagonal block A_{ii}. Assume that column 1 is fully summed and thus becomes a pivot column. First, we find the largest element, say element (2,1), in this column. Since row 2 belongs to the border, we cannot pick this as the pivot row. Now, find the largest element in column 1 restricting the search to rows that belong to the diagonal block, that is, rows 3–6. Say element (4,1) is the largest. Now, element (4,1) is

$$\begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & & \times & & & \times & & & \\
2 & \times & & & \times & & & & \\
\hline
3 & & \times & & \times & & & \times & \\
4 & \times & \times & & & & \times & & \\
5 & \times & & \times & & \times & & \times & \\
6 & \times & & \times & \times & & & & \times \\
\hline
\end{array}$$

Figure 1. Example used in text to explain pivoting strategy.

Rows 1–2 belong to the border, and rows 3–6 to the diagonal block.

chosen as the pivot if it satisfies a threshold stability criterion. That is, if element (4,1) has a magnitude greater than or equal to t times the magnitude of element (2,1), then element (4,1) is chosen as pivot, where t is a preset fraction (threshold pivot tolerance) in the range $0 < t \leq 1.0$. A typical value of t is 0.1. If a pivot search does not find an element that satisfies this partial-threshold criterion, then the elimination of that variable is delayed and the pivot column becomes part of the interface problem. If there are more than $n_i - m_i$ such delayed pivots, then $p_i < m_i$ and a row or rows of the diagonal block will also be made part of the interface problem. This has the effect of increasing the size of the interface problem; however, our computational experiments indicate that the increase in size is very small compared to n, the overall problem size.

Matrix reordering

As discussed previously, to make the solution method described earlier most effective, the size of the interface problem must be kept small. Furthermore, for load-balancing reasons, it is desirable that the diagonal blocks be nearly equal in size (and preferably that the number of them be a multiple of the number of processors to be used). For an ideal ordering, with each diagonal block presenting an equal workload and no interface matrix (i.e., a block-diagonal matrix), the speedup of the algorithm would in principle scale linearly. However, this ideal ordering rarely exists.

For a large-scale simulation or optimization problem, the natural unit-stream structure, as expressed in Eq. 4, may well provide an interface problem of reasonable size. However, load balancing is likely to be a problem, as the number of equations in different unit models may vary widely. This might be handled in an *ad hoc* fashion, by combining small units into larger diagonal blocks (with the advantage of reducing the size of the border) or by breaking larger units into smaller diagonal blocks (with the disadvantage of increasing the size of the border). Doing the latter also facilitates an equal distribution of the workload across the processors by reducing the granularity of the tasks. It should be noted in this context that in PFAMP task scheduling is done dynamically, with tasks assigned to processors as the processors become available. This helps reduce load imbalance problems for problems with a large number of diagonal blocks.

To address the issues of load balancing and of the size of the interface problem in a more systematic fashion, and to handle the situation in which the application code does not provide a bordered block-diagonal form directly in the first place, there is a need for matrix-reordering algorithms. For structurally *symmetric* matrices, there are various approaches that can be used to try to get an appropriate matrix reordering (e.g., Kernighan and Lin, 1970; Leiserson and Lewis, 1989; Karypis and Kumar, 1995). These are generally based on solving graph partitioning, bisection, or min-cut problems, often in the context of nested dissection applied to finite-element problems. Such methods can be applied to a structurally *asymmetric* matrix A by applying them to the structure of the symmetric matrix $A + A^T$, and this may provide satisfactory results if the degree of asymmetry is low. However, when the degree of asymmetry is high, as in the case of process simulation and optimization problems, the approach cannot be expected to always yield good results, as the number of additional nonzeros in $A + A^T$, indicating dependencies that are nonexistent in the problem, may be large, nearly as large as the number of nonzeros indicating actual dependencies.

To deal with structurally asymmetric problems, one technique that can be used is the min-net-cut (MNC) approach of Coon and Stadtherr (1995). This technique is designed specifically to address the issues of load balancing and interface problem size. It is based on recursive bisection of a bipartite graph model of the asymmetric matrix. Since a bipartite graph model is used, the algorithm can consider unsymmetric permutations of rows and columns while still providing a structurally stable reordering. The matrix form produced is a block-tridiagonal structure in which the off-diagonal blocks have relatively few nonzero columns; this is equivalent to a special case of the bordered block-diagonal form. The columns with nonzeros in the off-diagonal blocks are treated as belonging to the interface problem. Rows and other columns that cannot be eliminated for numerical reasons are assigned to the interface problem as a result of the pivoting strategy used in the frontal elimination of the diagonal blocks.

Another reordering technique that produces a potentially attractive structure is the tear_drop (tear, drag, reorder, partition) algorithm given by Abbott (1996). This makes use of the block structure of the underlying process-simulation problem, and also makes use of graph-bisection concepts. In this case, a doubly bordered block-diagonal form results. Rows and columns in the borders are immediately assigned to the interface problem, along with any rows and columns not eliminated for numerical reasons during factorization of the diagonal blocks.

In applying the frontal method to the factorization of each diagonal block, the performance may depend strongly on the ordering of the rows (but not the columns) within each block. No attempt was made to reorder the rows within each block to achieve better performance. It should be noted that the fill-reducing reordering strategies, such as minimum degree (Davis et al., 1996) for the symmetric case and Markowitz-type methods (Davis and Duff, 1997) for the unsymmetric case, normally associated with conventional multifrontal methods, are not directly relevant when the frontal method is used, though similar row-ordering heuristics can be developed for it (Camarda and Stadtherr, 1995).

In the computational results presented below, we make no attempt to compare different reordering approaches. The purpose of this article is to demonstrate the potential of the parallel frontal solver once an appropriate ordering has been established. It must be remembered, however, that the ordering used may have a substantial effect on the performance of the solver.

Results and Discussion

In this section, we present results for the performance of the PFAMP solver on three sets of process optimization or simulation problems. More information about each problem set is given below. We compare the performance of PFAMP on multiple processors with its performance on one processor and with the performance of the frontal solver FAMP on one processor. The numerical experiments were performed on a CRAY C90 parallel/vector supercomputer at Cray Research, Inc., in Eagan, Minnesota. The timing results presented represent the total time to obtain a solution vector from one righthand side vector, including analysis, factorization, and triangular solves. The time required to obtain an MNC or tear_drop reordering is not included. A threshold tolerance of $t = 0.1$ was used in PFAMP to maintain numerical stability, which was monitored using the 2-norm of the residual $b - Ax$. FAMP uses partial pivoting. While the times presented here are for the solution of a single linear system, it is important to realize that for a given dynamic simulation or optimization run, the linear system may need to be solved hundreds of times, and these simulation or optimization runs may themselves be done hundreds of times.

In the tables of matrix information presented below, each matrix is identified by name and order (n). In addition, statistics are given for the number of nonzeros (NZ) and for a measure of structural asymmetry (as). The asymmetry, as, is the number off-diagonal nonzeros a_{ij} ($j \neq i$) for which $a_{ji} = 0$ divided by the total number of off-diagonal nonzeros ($as = 0$ is a symmetric pattern, $as = 1$ is completely asymmetric). Also given is information about the bordered block-diagonal form used, namely the number of diagonal blocks (N), the order of the interface matrix (NI), and the number of equations in the largest and smallest diagonal blocks, $m_{i,\max}$ and $m_{i,\min}$, respectively.

Problem set 1

These three problems involve the optimization of an ethylene plant using NOVA, a chemical process optimization package from Dynamic Optimization Technology Products, Inc. NOVA uses an equation-based approach that requires the solution of a series of large sparse linear systems, which accounts for a large portion of the total computation time. The linear systems arising during optimization with NOVA are in bordered block-diagonal form, allowing the direct use of PFAMP for the solution of these systems. Matrix statistics for these problems are given in Table 1. Each problem involves a flowsheet that consists of 43 units, including five distillation columns. The problems differ in the number of stages in the distillation columns. Figure 2 presents timing results for the methods tested, with the parallel runs of PFAMP done on $P = 5$ processors. Not unsurprisingly the results of each problem are qualitatively similar, since all are based on the same underlying flowsheet.

Table 1. Description of Problem Set 1 Matrices*

Name	n	NZ	as	N	$m_{i,\max}$	$m_{i,\min}$	NI
Ethylene_1	10,673	80,904	0.99	43	3,337	1	708
Ethylene_2	10,353	78,004	0.99	43	3,017	1	698
Ethylene_3	10,033	75,045	0.99	43	2,697	1	708

*See text for definition of column headings.

We note first that the single processor performance of PFAMP is better than that of FAMP. This is due to the difference in the size of the largest frontal matrix associated with the frontal elimination for each method. For solution with FAMP, the variables that have occurrences in the border equations remain in the frontal matrix until the end. The size of the largest frontal matrix increases due to this reason, as does the number of wasted operations on zeros, thereby reducing the overall performance. This problem does not arise for solution with PFAMP because when the factorization of a diagonal block is complete, the remaining variables and equations in the front are immediately written out as part of the interface problem and a new front is begun for the next diagonal block. Thus, for these problems and most other problems tested, PFAMP is a more efficient single processor solver than FAMP, though both take good advantage of the machine-level parallelism provided by the internal architecture of the single processor. The usually better single-processor performance of PFAMP reflects the advantages of the multifrontal-type algorithm used by PFAMP, namely smaller and less sparse frontal matrices.

There are five large diagonal blocks in these matrices, corresponding to the distillation units, with one of these blocks much larger ($m_i = 3337$) than the others ($1185 \leq m_i \leq 1804$). In the computation, one processor ends up working on the largest block, while the remaining four processors finish the other large blocks and the several much smaller ones. The load is unbalanced with the factorization of the largest block becoming a bottleneck. This, together with the solution of the interface problem, another bottleneck, results in a speedup (relative to PFAMP on one processor) of less than

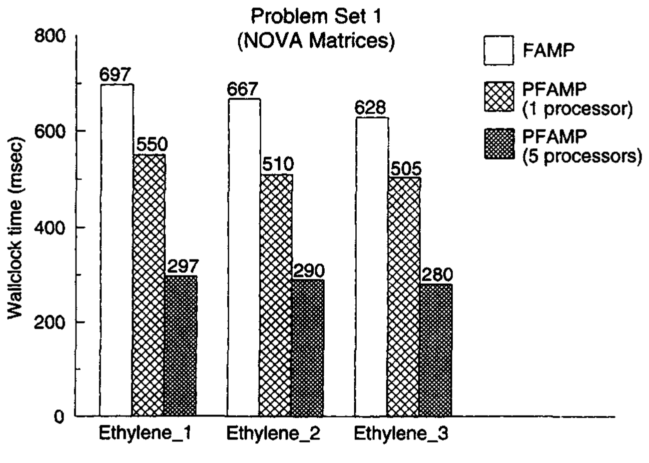

Figure 2. Comparison of FAMP and PFAMP on Problem Set 1 (NOVA matrices).

Table 2. Description of Problem Set 2 Matrices*

Name	n	NZ	as	N	$m_{i,\max}$	$m_{i,\min}$	NI
Hydr1c	5,308	23,752	0.99	4	1,449	1,282	180
Icomp	69,174	301,465	0.99	4	17,393	17,168	1,057
lhr_17k	17,576	381,975	0.99	6	4,301	1,586	581
lhr_34k	35,152	764,014	0.99	6	9,211	4,063	782
lhr_71k	70,304	1,528,092	0.99	10	9,215	4,063	1,495

*See text for definition of column headings.

two on five processors. It is likely that more efficient processor utilization could be obtained by using a better partition into bordered block-diagonal form.

Problem set 2

These five problems have all been reordered into a bordered block-diagonal form using the MNC approach (Coon and Stadtherr, 1995). Two of the problems (*Hydr1c* and *Icomp*) occur in dynamic simulation problems solved using SPEEDUP (Aspen Technology, Inc.). The *Hydr1c* problem involves a seven-component hydrocarbon process with a depropanizer and a debutanizer. The *Icomp* problem comes from a plantwide dynamic simulation of a plant that includes several interlinked distillation columns. The other three problems are derived from the prototype simulator SEQUEL (Zitney and Stadtherr, 1988), and are based on light hydrocarbon recovery plants, described by Zitney et al. (1996). Neither of the application codes produces directly a matrix in bordered block-diagonal form, so a reordering such as provided by MNC is required. Matrix statistics for these problems are given in Table 2. The SEQUEL matrices are more dense than the SPEEDUP matrices, and thus require a greater computational effort.

The performance of PFAMP on these problems is indicated in Figures 3 and 4. Since the performance of the frontal solver FAMP can depend strongly on the row ordering of the matrix, it was run using both the original ordering and the MNC ordering. For these problems, the difference in performance when using the different orderings was not significant. As in Problem Set 1, in most problems the PFAMP algorithm outperforms FAMP even on a single processor, for the reasons discussed before. This enhancement of performance can be quite significant, around a factor of 2 in the case of the SEQUEL matrices. MNC achieves the best reordering on the *Icomp* problem, for which it finds four diagonal blocks of roughly the same size ($17{,}168 \le m_i \le 17{,}393$) and the size of the interface problem is relatively small in comparison to n. The speedup observed for PFAMP on this problem was about 2.5 on four processors. While this represents a substantial savings in wallclock time, it still does not represent particularly efficient processor utilization. As noted earlier, the nearly serial performance on the interface problem, even though it is relatively small, can greatly reduce the overall efficiency of processor utilization.

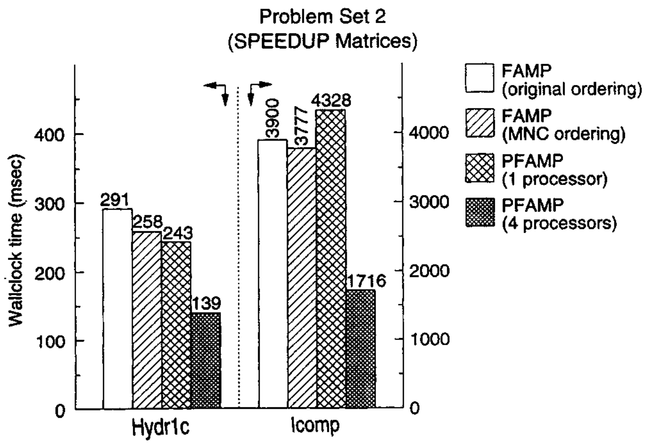

Figure 3. Comparison of FAMP and PFAMP on Problem Set 2 (SPEEDUP matrices).

Note that, as indicated by the arrows, the scale of the abscissa is not the same for both problems.

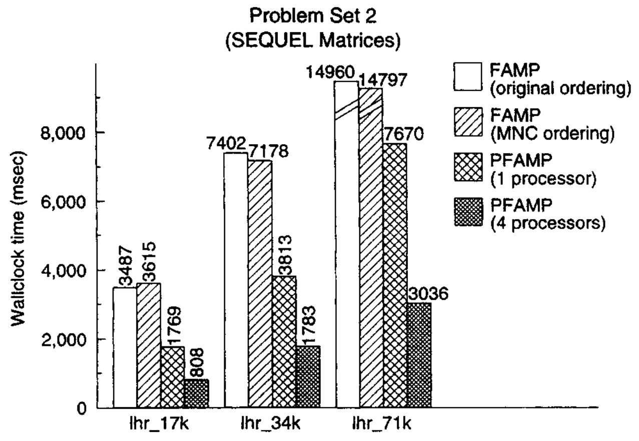

Figure 4. Comparison of FAMP and PFAMP on Problem Set 2 (SEQUEL matrices).

Problem set 3

These four problems arise from simulation problems solved using ASCEND (Piela et al., 1991), and ordered using the tear_drop approach (Abbott, 1996). Table 3 gives the statistics for each of the matrices. Problem *Bigequil.smms* represents a nine-component, 30-stage distillation column. Problem *Wood_7k* is a complex hydrocarbon separation process. Problems *4cols.smms* and *10cols.smms* involve nine components with four and ten interlinked distillation columns, respectively.

The performance of PFAMP is shown in Figure 5. On these problems, the moderate task granularity helps spread the load over the four processors used, but the size of the interface problem tends to be relatively large, 14–19% of n, as opposed to less than about 7% on the previous problems. The bets parallel efficiency was achieved on the largest problem (*10cols.smms*), with a speedup of about two on four proces-

Table 3. Description of Problem Set 3 Matrices*

Name	n	NZ	as	N	$m_{i,\max}$	$m_{i,\min}$	NI
Bigequil.smms	3,961	21,169	0.97	18	887	12	733
Wood_7k.smms	3,508	16,246	0.96	37	897	6	492
4cols.smms	11,770	43,668	0.99	24	1,183	33	2,210
10cols.smms	29,496	109,588	0.99	66	1,216	2	5,143

*See text for definition of column headings.

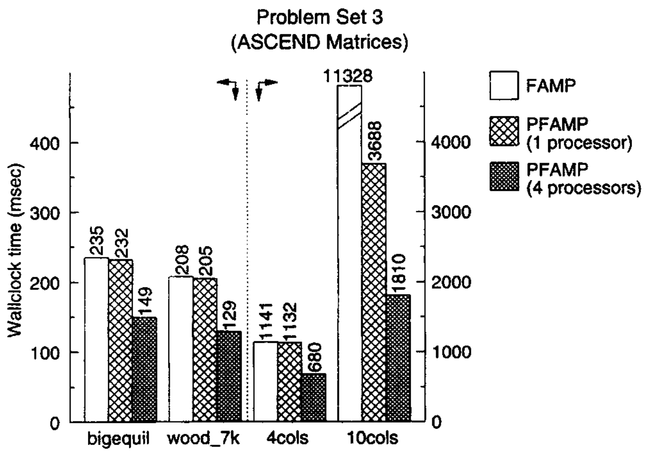

Figure 5. Comparison of FAMP and PFAMP on Problem Set 3 (ASCEND matrices).

Note that, as indicated by the arrows, the scale of the abscissa is not the same for all problems.

sors. This was achieved despite the relatively large size of the interface problem because, for this system, the use of small-grained parallelism within FAMP for solving the interface problem provided a significant speedup (about 1.7). As on the previous problems, this represents a substantial reduction in wallclock time, but is not especially good processor utilization. Overall on *10col.smms* the use of PFAMP resulted in the reduction of the wallclock time by a factor of 6; however, only a factor of 2 of this was due to multiprocessing.

Concluding Remarks

The results presented here demonstrate that PFAMP can be an effective solver for use in process simulation and optimization on parallel/vector supercomputers with a relatively small number of processors. In addition to making better use of multiprocessing than the standard solver FAMP, on most problems the single-processor performance of PFAMP was better than that of FAMP. The combination of these two effects led to four- to sixfold performance improvements on some large problems. Two keys to obtaining better parallel performance are improving the load balancing in factoring the diagonal blocks and better parallelizing the solution of the interface problem.

Clearly the performance of PFAMP with regard to multiprocessing depends strongly on the quality of the reordering into bordered block-diagonal form. In most cases considered earlier it is likely that the reordering used is far from optimal, and no attempt was made to find better orderings or compare the reordering approaches used. The graph-partitioning problems underlying the reordering algorithms are NP-complete. Thus, one can easily spend a substantial amount of computation time attempting to find improved orderings. The cost of a good ordering must be weighed against the number of times the given simulation or optimization problem is going to be solved. Typically, if the effort is made to develop a large-scale simulation or optimization model, then it is likely to be used a very large number of times, especially if it is used in an operations environment. In this case, the investment made to find a good ordering for PFAMP to exploit might have substantial long-term paybacks.

Acknowledgments

This work was supported by the National Science Foundation under Grants DMI-9322682 and DMI-9696110. We also acknowledge the support of the National Center for Supercomputing Applications at the University of Illinois, Cray Research, Inc., and Aspen Technology, Inc. We thank Kirk Abbott for providing the ASCEND matrices and the tear-drop reorderings.

Literature Cited

Abbott, K. A., "Very Large Scale Modeling," PhD Thesis, Dept. of Chemical Engineering, Carnegie Mellon Univ., Pittsburgh (1996).

Amestoy, P., and I. S. Duff, "Vectorization of a Multiprocessing Multifrontal Code," *Int. J. Supercomput. Appl.*, **3**, 41 (1989).

Camarda, K. V., and M. A. Stadtherr, "Exploiting Small-Grained Parallelism in Chemical Process Simulation on Massively Parallel Machines," AIChE Meeting, St. Louis, MO (1993).

Camarda, K. V., and M. A. Stadtherr, "Frontal Solvers for Process Simulation: Local Row Ordering Strategies," AIChE Meeting, Miami Beach, FL (1995).

Cofer, H. N., and M. A. Stadtherr, "Reliability of Iterative Linear Solvers in Chemical Process Simulation," *Comput. Chem. Eng.*, **20**, 1123 (1996).

Coon, A. B., and M. A. Stadtherr, "Generalized Block-Tridiagonal Matrix Orderings for Parallel Computation in Process Flowsheeting," *Comput. Chem. Eng.*, **19**, 787 (1995).

Davis, T. A., P. Amestoy, and I. S. Duff, "An Approximate Minimum Degree Ordering Algorithm," *SIAM J. Matrix Anal. Appl.*, **17**, 886 (1996).

Davis, T. A., and I. S. Duff, "An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization," *SIAM J. Matrix Anal. Appl.* (1997).

Duff, I. S., "Parallel Implementation of Multifrontal Schemes," *Parallel Comput.*, **3**, 193 (1986).

Duff, I. S., and J. R. Reid, "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Systems," *ACM Trans. Math. Softw.*, **9**, 302 (1983).

Duff, I. S., and J. R. Reid, "The Multifrontal Solution of Unsymmetric Sets of Linear Equations," *SIAM J. Sci. Stat. Comput.*, **5**, 633 (1984).

Duff, I. S., and J. A. Scott, "The Use of Multiple Fronts in Gaussian Elimination," Tech. Rep. RAL 94-040, Rutherford Appleton Laboratory, Oxon, UK (1994).

Hood, P., "Frontal Solution Program for Unsymmetric Matrices," *Int. J. Numer. Methods Eng.*, **10**, 379 (1976).

Ingle, N. K., and T. J. Mountziaris, "A Multifrontal Algorithm for the Solution of Large Systems of Equations Using Network-Based Parallel Computing," *Comput. Chem. Eng.*, **19**, 671 (1995).

Irons, B. M., "A Frontal Solution Program for Finite Element Analysis," *Int. J. Numer. Methods Eng.*, **2**, 5 (1970).

Karpis, G., and V. Kumar, "Multilevel *k*-way Partitioning Scheme for Irregular Graphs," Tech. Rep. 95-064, Dept. of Computer Science, Univ. of Minnesota, Minneapolis (1995).

Kernighan, B. W., and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.*, **49**, 291 (1970).

Leiserson, C. E., and J. G. Lewis, "Orderings for Parallel Sparse Symmetric Factorization," *Parallel Processing for Scientific Computing*, G. Rodrigue, ed., SIAM, Philadelphia, p. 27 (1989).

Lucas, R. F., T. Blank, and J. J. Tiemann, "A Parallel Solution Method for Large Sparse Systems of Equations," *IEEE Trans. Computed-Aided Design*, **CAD-6**, 981 (1987).

Mallya, J. U., "Vector and Parallel Algorithms for Chemical Process Simulation on Supercomputers," PhD Thesis, Dept. of Chemical Engineering, Univ. of Illinois, Urbana (1996).

Mallya, J. U., and M. A. Stadtherr, "A Multifrontal Approach for Simulating Equilibrium-Stage Processes on Supercomputers," *Ind. Eng. Chem. Res.*, **36**, 144 (1997).

Montry, G. R., R. E. Benner, and G. G. Weigand, "Concurrent Multifrontal Methods: Shared Memory, Cache and Frontwidth Issues," *Int. J. Supercomput. Appl.*, **1**(3), 26 (1987).

Piela, P. C., T. G. Epperly, K. M. Westerberg, and A. W. Westerberg, "ASCEND: An Object-oriented Computer Environment for Modeling and Analysis: The Modeling Language," *Comput. Chem. Eng.*, **15**, 53 (1991).

Stadtherr, M. A., and J. A. Vegeais, "Process Flowsheeting on Supercomputers," *Ind. Chem. Eng. Symp. Ser.*, **92**, 67 (1985).
Vegeais, J. A., and M. A. Stadtherr, "Vector Processing Strategies for Chemical Process Flowsheeting," *AIChE J.*, **36**, 1687 (1990).
Vegeais, J. A., and M. A. Stadtherr, "Parallel Processing Strategies for Chemical Process Flowsheeting," *AIChE J.*, **38**, 1399 (1992).
Westerberg, A. W., and T. J. Berna, "Decomposition of Very Large-Scale Newton-Raphson Based Flowsheeting Problems," *Comput. Chem. Eng.*, **2**, 61 (1978).
Zitney, S. E., "Sparse Matrix Methods for Chemical Process Separation Calculations on Supercomputers," *Proc. Supercomputing '92*, IEEE Press, Los Alamitos, CA, p. 414 (1992).
Zitney, S. E., L. Brüll, L. Lang, and R. Zeller, "Plantwide Dynamic Simulation on Supercomputers: Modeling a Bayer Distillation Process," *AIChE Symp. Ser.*, **91**(304), 313 (1995).
Zitney, S. E., J. U. Mallya, T. A. Davis, and M. A. Stadtherr, "Multifrontal vs. Frontal Techniques for Chemical Process Simulation on Supercomputers," *Comput. Chem. Eng.*, **20**, 641 (1996).
Zitney, S. E., and M. A. Stadtherr, "Computational Experiments in Equation-based Chemical Process Flowsheeting," *Comput. Chem. Eng.*, **12**, 1171 (1988).
Zitney, S. E., and M. A. Stadtherr, "Frontal Algorithms for Equation-based Chemical Process Flowsheeting on Vector and Parallel Computers," *Comput. Chem. Eng.*, **17**, 319 (1993).

Manuscript received Aug. 26, 1996, and revision received Dec. 20, 1996.